Appendix B

RAMでプログラムを動かす

山際伸一

組み込み機器は，利用者がスイッチをONすると自動的にプログラムを実行し始めるように設計しなければなりません．このため，起動時のソフトウェアはROMに配置します．しかし，起動後は，命令コードをRAMに配置させて動作させる機器もあります．　（筆者）

● 命令コードをRAMに配置する理由

パソコンのようにソフトウェアを入れ替えながら使うことがない組み込み機器で，なぜ命令コードをRAMに配置するのでしょうか．

いちばんの理由は，ROMのアクセス時間がRAMよりも長いというケースが多いことです．例えば，RAMなら0ウェイトで動作できるが，ROMでは1ウェイト以上必要になる場合があります．このときは，RAMを使ったほうが性能が向上します．

メモリのアクセス時間に差がなくてもRAMを用いたほうがよい場合もあります．本誌の付属基板に搭載されているADuC7000シリーズの場合，内蔵のROM（フラッシュ・メモリ）のデータ・バス幅は16ビットですが，RAMのデータ・バス幅は32ビットです．ARMプロセッサは，32ビット・アーキテクチャです．32ビット長の命令コードをフェッチするとき，ROMでは2クロック必要ですが，RAMなら1クロックですむということになります．このことから，ソフトウェアがRAMに配置されていたほうが高速に動作することがわかります．

● RAMで命令コードを動作させる手順

命令コードをRAMに配置させるためには，あらかじめROMに用意したコード群をRAMに転送します．この方法を**図1**に示します．

まず，電源投入またはリセット後，決められたアドレスから実行を始めます（**図1**の①）．ここはROM領域です．次にROM領域に書かれた命令コードをRAMに転送します（**図1**の②）．そして，RAM上の命令コードの入り口のアドレスに分岐します（**図1**の③）．

しくみとしては単純ですが，これを実現するソフトウェアを開発するときには，注意が必要です．

まず，ROMで動作するソフトウェアをそのままRAMに転送しても，正しく動作しないということです．ROMとRAMが配置されるアドレスは異なります．分岐命令のジャンプ先やデータの参照先は，RAMで動作するアドレスになっている必要があります．

例えば，ROMアドレス0x00000000から配置されるコードの中に0x00000014に分岐する命令があるとします．これはROM上で正しく動作するコードとします．これをRAMアドレス空間で動作させようと，アドレス0x00010000に転送したらどうなるでしょうか．正しく動作させるためには，この分岐先は0x00010014でなければなりません．0x00000014に分岐してしまったらROMに戻ってしまい，目的を達成できないことになります．

ROMには，RAMで動作することを考慮したコードを書き込んでおく必要があります．つまり，起動時にROMで動作するコードとRAMで実行するコードは，別々に用意する必要があるということです．このため，スタートアップ・ルーチンが2種類必要になります．

● ADuC7000シリーズの独特な方法

使用するマイコンによって独特な方法を用いなければならないこともあります．

ADuC7000シリーズの場合，リセットの直後は0x00000000

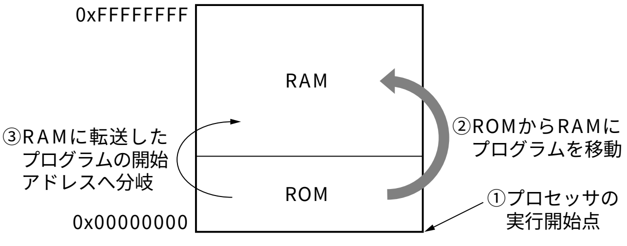


図1　RAMで動作するプログラムの初期の処理

開始アドレス	終了アドレス	領域
0xFFFF0000	0xFFFFFFFF	MMR
		予約
0x40000000	0x4000FFFF	外部メモリ領域3
		予約
0x30000000	0x3000FFFF	外部メモリ領域2
		予約
0x20000000	0x2000FFFF	外部メモリ領域1
		予約
0x10000000	0x1000FFFF	外部メモリ領域0
		予約
0x00080000	0x0008FFFF	フラッシュ・メモリ
		予約
0x00001000	0x00011FFF	SRAM
0x00000000	0x0000FFFF	再配置メモリ空間（フラッシュ・メモリまたはSRAM）

図2　メモリ・マップ

から命令の実行を始めます．また，REMAPレジスタの0ビット目に1を書くと，0x00000000がROMからRAMに変わります．REMAPレジスタへの書き込みは0x00008000以降の命令で行う必要がある，という制約があります．

このようなしくみでは，二つの異なるコードを同一アドレスに配置しなくてはならなくなるため，リンク時にエラーになってしまいます．

図2にADuC7000シリーズのメモリ・マップを示します．0x00080000からがROM（フラッシュ・メモリ）の領域です．0x00000000からの領域は，SRAMまたはフラッシュ・メモリの再配置空間です．つまり，0x00000000からの領域と0x00080000からの領域は，物理的には同一のメモリになり，同じ内容が見えるということになります．

そこで，以下の手順を実行するコードを記述すればよいことになります．

1）0x00000000から起動し，すぐに0x00080000へ分岐
2）REMAPレジスタに1を書く
3）RAMで実行する命令コードをROMからRAMにコピー
4）0x00000000に分岐

リンク時には，最初にROMから実行されるスタートアップ・ルーチンを0x00080000から配置します．スタートアップ・ルーチンの中には，0x00080000から領域に分岐する命令を記述して

▶ リスト1 startup2.s

```
    .text

    .extern      main
    .extern      _sp_base
#================================================
#      Initialize vectors
#
##     0x00      Reset
##     0x04      Undefined Instruction
##     0x08      Software Interrupt
##     0x0c      Abort(prefetch)
##     0x10      Abort(data)
##     0x14      Reserved
##     0x18      IRQ
##     0x1c      FIQ
#================================================

#   この_startupはアドレス0x00080000を保持する
    B    _startup

    .org(0x20)
_startup:
#   プログラムをROMからRAM領域にコピーする
#   再配置する際にはプログラムは0x00080000-0x0008FFFFで
#   動作していなければならない
    LDR  R1, =0xFFFF0220
    LDR  R2, =1
    STR  R2, [R1]
#   RAMで動作するためのプログラムのコピー
    LDR  R1, = _sstart_ram
    LDR  R2, = 0x0
    LDR  R3, = _end
copy_loop:
    LDR  R4, [R1]
    STR  R4, [R2]
    ADD  R1, R1, #4
    ADD  R2, R2, #4
    CMP  R1, R3
    bne  copy_loop


#   再配置後の0x00000000番地，つまりRAMの先頭に分岐
#   (リセット直後の状態を再現する)
    b    _sstart_ram
```

おきます．実際には，0x00000000から読み出されて実行することになりますが，0x00000000のアドレス空間から脱出できることになります．

● LED点滅のプログラムをRAMで実行

ここまで解説した方法を使って，第6章の演習2で説明したLED点滅のプログラムをRAM上で動作させてみます．

ここで使用するコードは，

1）リセット後にROMで動作するスタートアップ・ルーチン
2）RAMに分岐した後に実行するスタートアップ・ルーチン

です．これは，ROMとRAMにプログラムを配置するためのリンカ・スクリプトです．それぞれ順に見ていきます．

リスト1に，リセット後，ROMから実行されるコードを示します．アセンブリ言語で書かれています．**図3**に示すアルゴリズムで必要なコードをROMから読み出し，RAMに転送しています．ここで，`_sstart_ram`と`_end`はリンカ・スクリプトでアドレスが設定されます．つまり，`_sstart_ram`はRAMで実行されるコードの塊の先頭で，`_end`はその塊の最後の

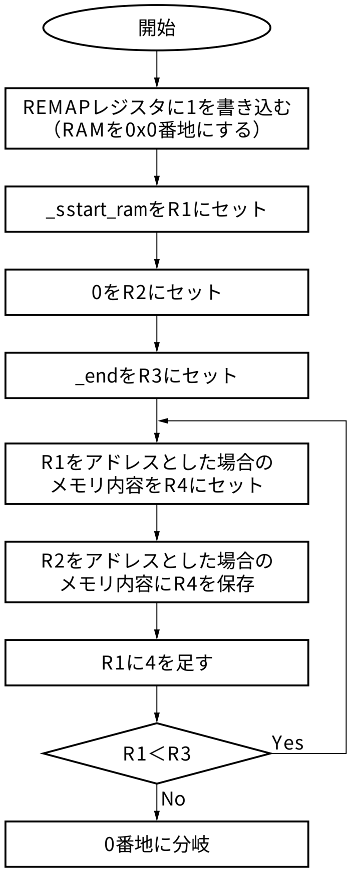


図3 プログラム転送アルゴリズム

◀ **リスト3 memory2.def**

```
SECTIONS
{
    .start 0x00080000 : {
    _sstart_rom = .;
        startup2.o(.text)
    _estart_rom = .;
    }
    .start_ram 0x0000000 : AT(LOADADDR(.start) + SIZEOF(.start)){
    _sstart_ram = LOADADDR(.start) + SIZEOF(.start);
        startup_ram.o
    _estart_ram = _sstart_ram + SIZEOF(.start_ram);
    }
    .text : AT(LOADADDR(.start_ram) + SIZEOF(.start_ram))
    {
    _stext = .;
        *(.text)
    _etext = .;
    }

    .rdata : AT(LOADADDR(.text) + SIZEOF(.text))
    {
    _srdata = .;
        *(.rodata)
            *(.rodata.str1.4)
    _erdata = .;
    }
    .data : AT(LOADADDR(.rdata) + SIZEOF(.rdata))
    {
    _sdata = .;
        *(.data)
        *(.zdata)
    _edata = .;
    }

    _end = LOADADDR(.data) + SIZEOF(.data);
    . += 0x100;
    _sp_base = .;

}
```

ROM上のアドレスを指しています．

RAMに分岐した後のスタートアップ・ルーチンを**リスト2**に示します．LEDを点滅させるコードは，第6章の演習2で使ったものと同じです．すなわち，RAMにコードを移した後は，今まで何もなかったかのように，実行するわけです．

▼**リスト2 startup_ram.s**

```
.text

    .extern       main
    .extern       _sp_base
#==================================================
#       Initialize vectors
#
##      0x00      Reset
##      0x04      Undefined Instruction
##      0x08      Software Interrupt
##      0x0c      Abort(prefetch)
##      0x10      Abort(data)
##      0x14      Reserved
##      0x18      IRQ
##      0x1c      FIQ
#==================================================

    B    _startup_ram
    .org(0x20)
_startup_ram:
    LDR  R13,=_sp_base
    BL   main
```

● コンパイルと実行

GCCを使うコンパイル手順を以下に示します．コンパイルの際には，メモリ配置を定義するmemory2.def（**リスト3**）というファイルを使用します．

```
$ arm-elf-gcc -c practice2.c
$ arm-elf-as startup2. s -o startup2.o
$ arm-elf-as startup_ram.s -o startup_
                                    ram.o
$ arm-elf-ld -T memory2.def startup2.o
startup_ram.o practice2.o -Map practice3.
                         map -o practice3
$ arm-elf-objcopy -O ihex practice3
                              practice3.hex
```

動作させてみると，点滅が速くなることがわかります．LEDのON/OFFを切り替えているmain部分は第6章の演習2と同じソース・コードなので，それだけ性能が向上したということになります．

LEDのON/OFFを切り替える待ち時間のループ回数であるITERATIONの値を大きくしていくと，徐々に演習2のときと同じ間隔に近づいていくので試してみてください．

やまぎわ・しんいち　工学博士